SOFTWARE WATERMARKING TECHNIQUES

FIELD OF THE INVENTION

The present invention relates to methods for protecting software against theft, establishing/proving ownership of software and validating software. More particularly, although not exclusively, the present invention provides for methods for watermarking what will be generically referred to as software objects. In this context, software objects may be understood to include programs and certain types of media.

BACKGROUND TO THE INVENTION

Watermarking is the process of embedding a secret message, the *watermark*, into a cover or overt message. For example, in media watermarking, the secret is commonly a copyright notice and the cover is a digital image, video or audio recording. Fingerprinting is a method whereby each individual software application incorporates a, potentially, unique, watermark which allows that particular example of the software to be identified. Fingerprinting may be viewed as a multiple use of watermarking techniques.

The watermark is constructed to make it difficult to remove the watermark without damaging the software object in which it is embedded. Such watermarks may only be removed safely by someone (or some process) in possession of one or more secrets that were employed while constructing the watermark.

Watermarking a software object (hereafter referred to as an *object*) discourages intellectual property theft. A further application is that watermarking an object can be used to establish and/or prove evidence of ownership of an object. Fingerprinting is similar to watermarking except a different watermark is embedded in every cover message thus providing a unique fingerprint for every object. Watermarking is therefore a subset of fingerprinting and the latter may be used to detect not only the fact that a theft has occurred, but may also allow identification of the particular object and thus establish an audit trail which can be used to reveal the infringer of copyright.

In the context of prior art watermark techniques, the following scenario serves to illustrate the ways in which a watermarked object may be vulnerable to attack. With

reference to figure 1, suppose that A watermarks an object *O* with a watermark *W* and key *K*. If the object *O* is sold to B and B wishes to (illegally) on-sell *O* to C, there are various types of attack to which *O* may be vulnerable.

Detection: initially B must try and detect the presence of the watermark in *O*. If there is no watermark, no further action is necessary.

Locate and remove: once B has determined that *O* carries a watermark, B may try to locate and remove *W* without otherwise harming the rest of the contents of O.

Distort: if some degradation in quality of O is acceptable, B may distort it sufficiently so that it becomes impossible for A to detect the presence of the watermark *W* in the object *O*.

Add: alternatively, if removing the watermark *W* is too difficult, or distorting the object *O* is not acceptable, B might simply add his own watermark *W'* (or several such marks) to the object *O*. This way, A's mark becomes just one of many.

It is considered that most media watermarking schemes are vulnerable to attack by distortion. For example, image transforms such as cropping and lossy compression will distort the image sufficiently to render many watermarks unrecoverable.

To the knowledge of the applicants there exists no effective watermarking scheme which is capable of use with or appropriate for software. It would be a significant advantage to be able to apply watermarking techniques to software in view of the widespread occurrence of software piracy. It is estimated at software piracy costs approximately 15 billion dollars per year. Thus the problem of software security and protection is of significant commercial importance.

One simple way, known in the prior art, of embedding a watermark in a piece of software is simply to include it in the initialized static data section of the object code. In a similar, yet more complex manner, watermarks are often encoded in what is known as an "Easter egg". This is a piece of code, which is activated for a highly unusual or seldom encountered input to the particular application, which displays a

watermark image, plays a watermark sound, or, in some way, alerts the user that the watermark code has been activated.

Thus, it is an object of the present invention to provide methods for watermarking software objects which overcomes the limitations inherent in prior art watermarking techniques and allows for non-media objects to be watermarked effectively. It is a further object of the present invention to provide methods for watermarking software objects which are resistant to the aforementioned techniques for attacking watermark objects or to at least provide the public with a useful choice.

DISCLOSURE OF THE INVENTION

In one aspect, the invention provides for a method of watermarking a software object whereby a watermark is stored in the state of the software object as it is being run with a particular input sequence.

The software object may be a program or piece of program.

The state of the software object may correspond to the current values held in the stack, heap, global variables, registers, program counter and the like.

In a preferred embodiment, the watermark may be stored in an object's execution state whereby a (possibly empty) input sequence *I* is constructed which, when fed to an application of which the object is a part, will make the object *O* enter a state which represents the watermark, the representation being validated or checked by examining the dynamically allocated data structures of the object *O*.

In an alternative embodiment, the watermark could be embedded in the execution trace of the object *O* whereby, as a special input *I* is fed to *O*, the address/operator trace is monitored and, based on a property of the trace, a watermark is extracted.

In a preferred embodiment, the watermark is embedded in the state of the program as it is being run with a particular input sequence $I=I_1 \ldots I_k$.

The watermark may be embedded in the topology of a dynamically built graph structure.

The graph structure (or watermark graph) corresponds to a representation of the data structure of the program and may be viewed as a set of nodes together with a set of vertices.

The method may further comprise building a recognizer *R* concurrently with the input *I* and watermark *W*.

Preferably *R* is a function adapted to identify and extract the watermark graph from all other dynamically allocated data structures.

In an alternative, less preferred embodiment, the watermark *W* may incorporate a marker that will allow *R* to recognize it easily.

In a preferred embodiment, *R* is retained separately from the program whereby *R* is dynamically linked with the program when it is checked for the existence of a watermark.

Preferably the application of which the object forms a part is obfuscated or incorporates tamper-proofing code.

In a preferred embodiment, *R* extracts a value *n* from the topology of the graph comprising the watermark *W*.

The watermark *W* has a signature property *s* where *s(W)* evaluates to "true" if the watermark *W* is recognisable wherein the recogniser *R* tests a presumed watermark *W'* by evaluating the signature property *s(W')*.

In a preferred embodiment, the method includes the creation of a number *n* which may be embedded in the topology of a watermark graph, wherein the signature property *s(W)* is a function of a number n so embedded.

In a preferred embodiment, the signature property *s(W)* is "true" if and only if the number *n* is the product of two primes.

The invention further provides for a method of verifying the integrity or origin of a program including:

watermarking the program with a watermark *W* in the state of a program as the program is being run with a particular input sequence *I*;

building a recognizer *R* concurrently with the input *I* and watermark *W* wherein the recognizer is adapted to extract the watermark graph from other dynamically allocated data structures wherein *R* is kept separately from the program; wherein *R* is adapted to check for a number *n*, *n*, in a preferred embodiment, being the product of two primes and wherein *n* is embedded in the topology of *W*.

Preferably, the signature property may be evaluated by testing for a specific result from a hard computational problem.

The number *n* may be derived from any combination of numbers depending on the context and application.

Preferably the program or code is further adapted to be resistant to tampering, preferably by means of obfuscation or by adding tamper-proofing code.

Preferably the watermarks *W* are chosen from a class of graphs *G* wherein each member of *G* has one or more properties, such as planarity, said property being capable of being tested by integrity-testing software.

In an alternative embodiment, the watermark may be rendered tamperproof to certain transformations, such as attacks, by expanding each node of the watermark graph into a *j*-cycle, where *j* may be any number from 1 to 5.

In a broad aspect, the recognizer *R* checks for the effect of the watermarking code on the execution state of the application thereby preserving the ability to recognize the watermark in cases where semantics-preserving transformations have been applied to the application.

In a further aspect, the invention provides for a method of watermarking software including the steps of:

embedding a watermark in a static string, then applying an obfuscation technique whereby this static string is converted into executable code.

The executable code is called whenever the static string is required by the program.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will now be described by way of example only and with reference to the figures in which:

Figure 1: illustrates methods of adding a watermark to an object and attacking the integrity of such a watermark;

Figure 2: illustrates methods of embedding a watermark in a program;

Figure 3: illustrates an example of a function used to embed a watermark within a static string;

Figure 4: illustrates insertion of a bogus predicate into a program;

Figure 5: illustrates splitting variables;

Figure 6: illustrates merging variables;

Figure 7: illustrates the conversion of a code section into a different virtual machine code;

Figure 8: illustrates an example of a method of the watermarking scheme according to the present invention;

Figure 9: illustrates a possible encoding method for embedding a number in the topology of a graph;

Figure 10: illustrates another possible embodiment for embedding a number in the topology of a graph;

Figure 11; illustrates a marker in a graph;

Figure 12: illustrates examples of obfuscating transformations;

Figure 13: illustrates examples of tamperproofing Java code;

Figure 14: illustrates enumeration encoding in a planted plane cubic tree on $2m$ = 8 nodes; and

Figure 15: illustrates tamperproofing against node-splitting.

Referring to Figure 1(b) a way is shown by which Bob can circumvent a watermarking scheme by distorting the protected object. If the distortion is at ``just the right level'', *O* will still be usable by Bob, but Charles will be unable to extract the watermark. In Figure 1(9), the distortion is so severe that *O* is no longer functional, so Bob will not be able to use it, nor is he able to on-sell it.

In the present context, tamperproofing is applied in order to prevent an adversary from removing the watermark and to provide assurance to the software end-user that the software object hasn't been tampered with. Thus the 'integrity' of the program may be verified. The primary aim of the present invention is to allow accurate assertion of ownership of a software object with a secondary purpose being to ensure the integrity of the object.

It has been shown that there are transformations, called obfuscating transformations, that will destroy almost any kind of program structure while preserving the semantics (operational behaviour) of the program. Other semantics preserving transformations, such as optimising transformations known from the prior art can be used to similar effect. As a consequence, any software watermarking technique must be evaluated with respect to its resilience to attack from automatic application of semantics preserving transformations, such as obfuscation. The following discussion will survey obfuscating transformations that can be used to destroy software watermarks.

In Figure 2a a watermark is embedded within a static string. There are several ways of rendering watermarks unrecogisable, the most effective perhaps by converting static strings into a program that produces the data. As an example, consider the function G in Figure 3. This function was constructed to obfuscate the strings "AAA", "BAAAA", and "CCB". The values produced by G are G(1)="AAA", G(2)="BAAAA", G(3)=G(5)="CCB", and G(4)="XCB".

In Figure 2b Alice embeds a watermark within the program code itself. There are numerous ways to attack such code. Figure 4, for example, shows how it is possible to insert bogus predicates into a program. These predicates are called opaque since their outcome is known at obfuscation time, but difficult to deduce otherwise. Highly resilient opaque predicates can be constructed using hard static analysis problems such as aliasing.

In Figure 2c a watermark is embedded within the state (global, heap, and stack data, etc.) of the program as it is being run with a particular input *I*. Different obfuscation techniques can be employed to destroy this state, depending on the type of the data. For example, one variable can be split into several variables (Figure 5) or several variables can be merged into one (Figure 6).

In Figure 2d a watermark is embedded within the trace (either instructions or addresses, or both) of the program as it is being run with a special input sequence $I = I_1, I_2, \ldots I_k$. In an alternative embodiment, a watermark may be embedded within a series of execution traces, said series of traces being generated as the program is run on a special input. This special input is comprised of a series of one or more input sequences, where each input sequence is generated by a specific process

which may incorporate a random or pseudorandom number generator. Execution traces have many properties that may be observed by a watermark recogniser *R*. One example of such a property is "if the program passes point P1 in O, then there's a 32% chance that it will also pass point P2". Another example of such a property is the frequency at which some specific basic operation, such as addition, is performed. A specific collection of (one or more) such execution-trace properties is the watermark *W*. The signature property *s(W)* for this *W* is that all the property values are within some predefined tolerance. For example, we might require that our sample property P1-P2 have a value between 30% and 34% on a randomly-generated series of 10000 inputs (note that we would not expect to observe an "exact match" to our 32% estimated mean-value for this property P1-P2, because each randomly-generated series of inputs would give us a somewhat different measurement for this property value).

Many of the same transformations that can be used to obfuscate code will also obfuscate an instruction trace. Figure 7 shows another, more potent, transformation. The idea is to convert a section of code (Java bytecode in our case) into a different virtual machine code. The new code is then executed by a virtual machine interpreter included with the obfuscated application. The execution trace of the new virtual machine running the obfuscated program will be completely different from that of the original program. In Figure 2e, a watermark is embedded in an Easter Egg. Unless the code is obfuscated, Easter Eggs may be found by straightforward techniques such as decompilation and disassembly.

In this section, techniques for embedding software watermarks in dynamic data structures are discussed. The inventors view these techniques as the most promising for withstanding de-watermarking attacks by obfuscation.

The basic structure of the proposed watermarking technique is outlined in Figure 8. The method is as follows:

1. The watermark *W* is embedded, not in the static structure of the program, its code (Unix text segment), its static data (Unix initialised data segment), or its type information (Unix symbol segment or Java's Constant Pool), but rather in the state of the program as it is being run with a particular input sequence *I*

(of length k) whose elements are $I = I_1, I_2 \ldots I_k$. Of course k may be 0, in which case there is no input and the input sequence is empty.

2. More specifically, the watermark is embedded in the topology of a dynamically built graph structure. It is believed that obfuscating the topology of a graph is fundamentally more difficult than obfuscating other types of data. Moreover, it is anticipated that tamperproofing such a structure should be easier than tamperproofing code or static data. This is particularly true of languages like Java, where a program has no direct access to its own code.

3. A Recogniser R is built along with the input I and watermark W. R is a function that is able to identify and extract the watermark graph from among all other dynamic allocated data structures. Since, in general, sub-graph isomorphism is a difficult problem, it is possible that W will have some special marker that will allow R to recognise W easily. Alternatively, W may be formed immediately after input I_k is processed, i.e. markers may not be necessary. Markers are considered 'unstealthy' for the following reason. If a marker is easily recognisable by a recogniser, an adversary might discover it – perhaps by way of a collusive attack on a collection of fingerprinted objects. The use of markers can be avoided by exploiting the recogniser's knowledge of the secret input sequence in the following way: the watermark will be completed immediately after the k^{th} input (I_k) of this sequence is presented to the program. The recogniser knows the value of "k" and therefore is able to look for the watermark graph effectively, by examining the nodes that were allocated or modified during the processing of I_k. In contrast, the adversary would be unaware of the length of this sequence and would therefore have to "guess' a value of "k" as well as the values (I_1, $I_2 \ldots I_k$) in the input sequence I, before looking for the watermark.

4. An important aspect of the proposed technique is that R is not distributed along with the rest of the program. If it were, a potential adversary could identify and decompile it, and discover the relevant property of W. R is employed only when we check for the watermark. R may be an extension of the program comprised of self-monitoring code, or it may be an adjunct to a debugger or some other external means for examining the dynamic state of

the program. *R* may be linked in dynamically with the program when we check for the watermark. Other mechanisms are envisaged by which the recogniser *R* may observe the state of the object *O*.

5. It is required that some signature property *s(W)* of *W* be highly resilient to tampering. This can be achieved, for example, by obfuscation or by adding tamperproofing code to the application.

6. In Figure 8 it is assumed that the signature that *R* checks for is a number *n*, which has been embedded in the topology of *W*. *n* is the product of two large primes *P* and *Q*. To prove the legal origin of the program, we link in *R*, run the resulting program with *I* as input, and show that we can factor the number that *R* produces. Alternatively, s(*W*) can be based on hard computational problems other than factorisation of large integers.

The above issues will now be discussed in more detail. The first problem to be solved is how to embed a number in the topology of a graph. There are a number of ways of doing this, and, in fact, a watermarking tool should have a library of many such techniques to choose from. Figure 9 illustrates one possible encoding. The structure is basically a linked list with an extra pointer field which encodes a base-6 digit. A null-pointer encodes a 0, a self-pointer a 0, a pointer to the next node encodes a 1, etc. A further example is shown in figure 14 whereby the watermark *W* is chosen from a class of graphs *G* wherein each member of *G* has one or more properties (in figure 14 – planarity) that may be tested by integrity-checking software. The integrity checking software may be incorporated into the program during the watermarking process.

In the previous paragraph, it was shown how an integer *n* could be encoded in the topology of a graph. The encoding is resilient to tampering, as long as the recogniser *R* is able to correctly identify the nodes containing the two pointer fields in which we have encoded *n*. We now describe another encoding showing that a recogniser *R* can evaluate *n* if it can identify only a single pointer field per node.

Using a single pointer per node, we can construct a watermark *W* in the form of a parent-pointer tree. The parent-pointer tree *W* is a representation of a graph *G*

known as an oriented tree enumerable by the techniques described in Knuth, Vol I 3rd Edition, Section 2.3.4.4.

The number a_m of oriented trees with m nodes is asymptotically $a_m = c(1/\alpha)^{n-1}/n^{3/2} + O((1/\alpha)^n/n^{5/2})$ for $c \sim 0.44$ and $1/\alpha \sim 2.956$. Thus we can encode an arbitrary 1000-bit integer n in a graphic watermark W with $1000/\log_2 2.956 \sim 640$ nodes.

We construct an index n for any enumerable graph in the usual way, that is, by ordering the operations in the enumeration. For example, we might index the trees with m nodes in ``largest subtree first" order, in which case the path of length $m-1$ would be assigned index 1. Indices 2 through a_{m-1} would be assigned to the other trees in which there is a single subtree connected to the root node. Indices $a_{m-1}+1$ through $a_{m-1} + a_{m-2}$ would be assigned to the trees with exactly two subtrees connected to the root node, such that one of the subtrees has exactly $m-2$ nodes. The next $a_{m-3}a_2 = a_{m-1}$ indices would be assigned to trees with exactly two subtrees connected to the root node, such that one of the subtrees has exactly $m-3$ nodes. See Figure 10 for an example.

To aid the recognition of a watermark, the recogniser may use secret knowledge of a "signal" indicating that "the next thing that follows" is the real watermark. In a preferred embodiment, the secret is the input sequence I; the recogniser (but not the attacker) knows that the watermark will be constructed after the input sequence $I = I_1, I_2 \ldots I_k$ has been processed. In an alternative, but less preferred embodiment, the secret is an easily recognisable "marker" that may be present in the watermark graph. This is similar to the signals used between baseball coaches and their players. See Figure 11 for an example.

One advantageous consequence of the present approach is that semantics-preserving transformations, such as those employed in optimising compilers and those employed by obfuscation techniques which target code and static data will have no effect on the dynamic structures that are being built. There are, however, other techniques which can obfuscate dynamic data, and which we will need to tamperproof against. There are three types of obfuscating transformations which will need to be protected against:

1. An adversary can add extra pointers to the nodes of linked structures. This will make it hard for *R* to recognise the real graph within a lot of extra bogus pointer fields.

2. An adversary can rename and reorder the fields in the node, again making it hard to recognise the real watermark.

3. Finally, an adversary can add levels of indirection, for example by splitting nodes into several linked parts.

These transformations are illustrated in Figure 12. It is important to note here that obfuscating linked structures has some potentially serious consequences. For example, splitting nodes will increase the dynamic memory requirement of the program (each cell carries a certain amount of overhead for type information etc.), which could mean that a program which ran on, say, a machine with 32M of memory would now not run at all. Furthermore, if we assume that an adversary does not know in which dynamic structure our watermark is hidden, he is going to have to obfuscate every dynamic memory allocation in the entire program.

Next will be discussed techniques for tamperproofing a dynamic watermark against the obfuscation attacks outlined above.

The types of tamperproofing techniques that will be available will depend on the nature of the distributed object code. If the code is strongly typed and supports reflection (as is the case with Java bytecode) we can use these reflection capabilities to construct the tamperproofing code. If, on the other hand, the application is shipped as stripped, untyped, native code (as is the case with most programs written in C, for example) this possibility is not open to us. Instead, we can insert code which manipulates the dynamically allocated structures in such a way that obfuscating them would be unsafe.

ANSI C's address manipulation facilities and limited reflection capabilities allow for some trivial tamperproofing checks:

```
include <stdlib.h>
```

```
include <stddef.h>
struct s int a; int b;;
void main ()
        if (offsetof(struct s, a) >
                offsetof(struct s, b)) die();
        if (sizeof(struct s) != 8) die();
}
```

These tests will cause the program to terminate if the fields of the structure are reordered, or the structure is split or augmented.

Figure 13 (a) shows how Java's reflection package allows us to perform similar tamperproofing checks. Note that this example code is not completely general, since Java does not specify the relative order of class fields.

Figure 13 (b) shows how we can also use opaque predicates and variables to construct code which appears to (but in fact, does not) perform ``unsafe" operations on graph nodes. A de-watermarking tool will not be able to statically determine whether it is safe to apply optimising or obfuscating transformations on the code. In the example in Figure 13 (b), V is an opaque string variable whose value is "car", although this is difficult for a de-watermarker to work out statically. At 1 it appears as if some or all (unknown to the de-watermarker) field is being set to null, although this will never happen. The statement 2 is a redundant operation performing n.car = n.car, although (due to the opaque variable R whose value is always 1) this cannot in general be worked out statically.

For increased obscurity, the code to build the watermark should be scattered over the entire application. The only restriction is that when the end of the input sequence $I=I_1 \ldots I_k$ is reached, the watermark W has been constructed. This watermark in a preferred embodiment, may be composed of some or all of the *components* $W_1, \ldots W_{k-1}$ that were constructed previously. Additionally, in a preferred embodiment, some components W_i may be composed of some of all components constructed before W_i.

W_0 =...;
if (input = I_1) W_1 =...;
if (input = I_2) W_2 = ...;

if (input = I_{k-1}) W_{k-1} = ...;
if (input = I_k) W = ...;

In order to identify the watermark structure, the recogniser must be able to enumerate all dynamically allocated data structures. If this is not directly supported by the runtime environment (as, for example, is the case with Java), we have two choices. We can either rewrite the runtime system to give us the necessary functionality or we can provide our own memory allocator. Notice, though, that this is only necessary when we are attempting to recognise the watermark. Under normal circumstances the application can run on the standard runtime system.

A further technique is shown in figure 15. Here is illustrated a technique which applies a local transformation, thereby tamperproofing the watermark against an attack by node-splitting. Each of the nodes of the original watermark graph is expanded into a 4-cycle. If an adversary splits two nodes, the underlying structure ensures that these node will fall on a cycle. At (3) the recogniser shrinks the biconnected components of the underlying graphs with the result that the graph is isomorphic to the original watermark.

It is envisaged that local transformations, other than expansion of nodes into cycles, may be employed to tamperproof the watermark against specific attackes other than node-splitting. For example, redundant edges may be introduced into the watermark in order to render the watermark tamperproof to specific attacks which involve the renaming and reordering of fields in nodes.

A number of techniques are known in the prior art for hiding copyright notices in the object code of a program. It is the inventors' belief that such methods are not resilient to attack by obfuscation – an adversary can apply a series of transformations that will hide or obscure the watermark to the extent that it can no longer be reliably retrieved.

The present invention indicates that the most reliable place to hide a watermark is within the dynamically allocated data structures of the program, as it is being executed with a particular input sequence.

A further application for the watermarking technique described above may be in "fingerprinting" software. In this case, each individual program (i.e. every distributed copy of the code) is watermarked with a different watermark. Although there is a risk of an adversary collusively attacking the watermark, the applicant believes that applying obfuscation may render it very difficult for the attacker to interpret the evidence obtained by a collusive attack.

Where in the foregoing description reference has been made to elements or integers having known equivalents, then such equivalents are included as if they were individually set forth.

Although the invention has been described by way of example and with reference to particular embodiments, it is to be understood that modifications and/or improvements may be made without departing from the scope or spirit of the invention.